Remote State Management

Terraform Day2 – module 6

Learning Objectives

By the end of this module, you will be able to:

- Explain why local Terraform state is not suitable for team environments
- Describe how remote backends enable collaboration and reliability
- Configure AWS S3 as a Terraform remote backend
- Explain S3 native state locking using versioning
- Migrate safely from local state to remote state
- Apply security best practices for Terraform state files
- Use terraform_remote_state to share data across projects

Recap: Local State Challenges

What is Terraform State?

- terraform.tfstate:
- Tracks real infrastructure
- Stores resource IDs and attributes
- Maintains dependency relationships
- Is essential for Terraform to function correctly

Thought Question:

👉 *What happens if two engineers run* `terraform apply` *at the same time on their laptops?*

Problems with Local State:

- Stored only on your laptop
- May contain **sensitive data** (secrets, private IPs, DB passwords)
- No built-in locking mechanism
- Cannot be easily shared with teammates

The Problem: Local State in Teams

Scenario: Bank Payment System

Alice (Developer A)

```
terraform apply
```

- ➡️ Creates aws_instance.payment_server
  ➡️ State saved only on Alice's laptop

Bob (Developer B)

```
terraform apply
```

➡️ Creates another `aws_instance.payment_server`
➡️ State saved only on Bob's laptop

Result:

- Two servers created instead of one
- Two different state files
- No "single source of truth"
- Manual cleanup required

🚨 **This is a critical production risk.**

Remote State: The Solution

What is Remote State?

- Remote state stores Terraform state in a **shared location** such as AWS S3.

Architecture:

- Alice, Bob, Jenkins → S3 Remote State

Benefits (Stronger Wording):

- **Single source of truth**
- **State locking** (prevents parallel applies)
- **Encryption** (protects secrets at rest)
- **Versioning** (rollback to previous state)
- **Access control via IAM**

Why AWS S3 for Terraform State?

S3 is ideal because it provides:

- **High durability (11 9's)**
- **Built-in versioning**
- **Server-side encryption (SSE)**
- **Fine-grained IAM access control**
- **Low cost**
- **Native state locking using versioning (No DynamoDB required)**

Architecture: Remote State Workflow

Terraform Workflow:

- terraform init → Configure S3 backend
- terraform plan
  - Acquires lock
  - Reads latest state
  - Generates plan
  - Releases lock
- terraform apply
  - Acquires lock
  - Applies changes
  - Writes new state version to S3
  - Releases lock

Security Controls:

- S3 encryption enabled
- Versioning enabled
- IAM least privilege

Step 1: Create S3 Bucket

Your S3 bucket must have:

- ✅ Versioning enabled
- ✅ Encryption enabled
- ✅ Block public access
- ✅ Globally unique name

Why versioning matters:

- Enables native state locking
- Keeps history of all changes
- Helps recovery in case of mistakes

Step 2: Bootstrap Setup (Chicken-Egg Problem)

Problem:

- You need S3 to store state, but S3 itself must be created using Terraform.

Solution (Bootstrap approach):

- Create a separate "bootstrap" Terraform project
- Use local state **only once** to create the S3 bucket
- After that, all projects use remote state

Step 3: Configure S3 Backend

```
terraform {
  backend "s3" {
    bucket = "myorg-terraform-state"
    key    = "prod/app/terraform.tfstate"
    region = "us-east-1"
    encrypt = true
    use_lockfile = true
  }
}
```

Key Parameters:

- `bucket` → Your state bucket
- `key` → Organize by project/environment
- `use_lockfile = true` → Enables S3 native locking

Step 4: Migrate to Remote State

Run:

terraform init -migrate-state

Terraform will:

- Detect new backend
- Ask for confirmation
- Copy local state to S3
- Switch all future operations to remote state

After Migration:

- State lives in S3
- Local .tfstate can be deleted

State Locking in Action

- When you run:

  terraform apply

- Terraform displays:

  Acquiring state lock…

- If another user tries to apply:

```
Error: Error acquiring state lock
Lock Info:
  Who: alice@example.com
  Operation: apply
```

Why this matters:

- Prevents accidental parallel changes
- Protects infrastructure integrity

Force Unlock

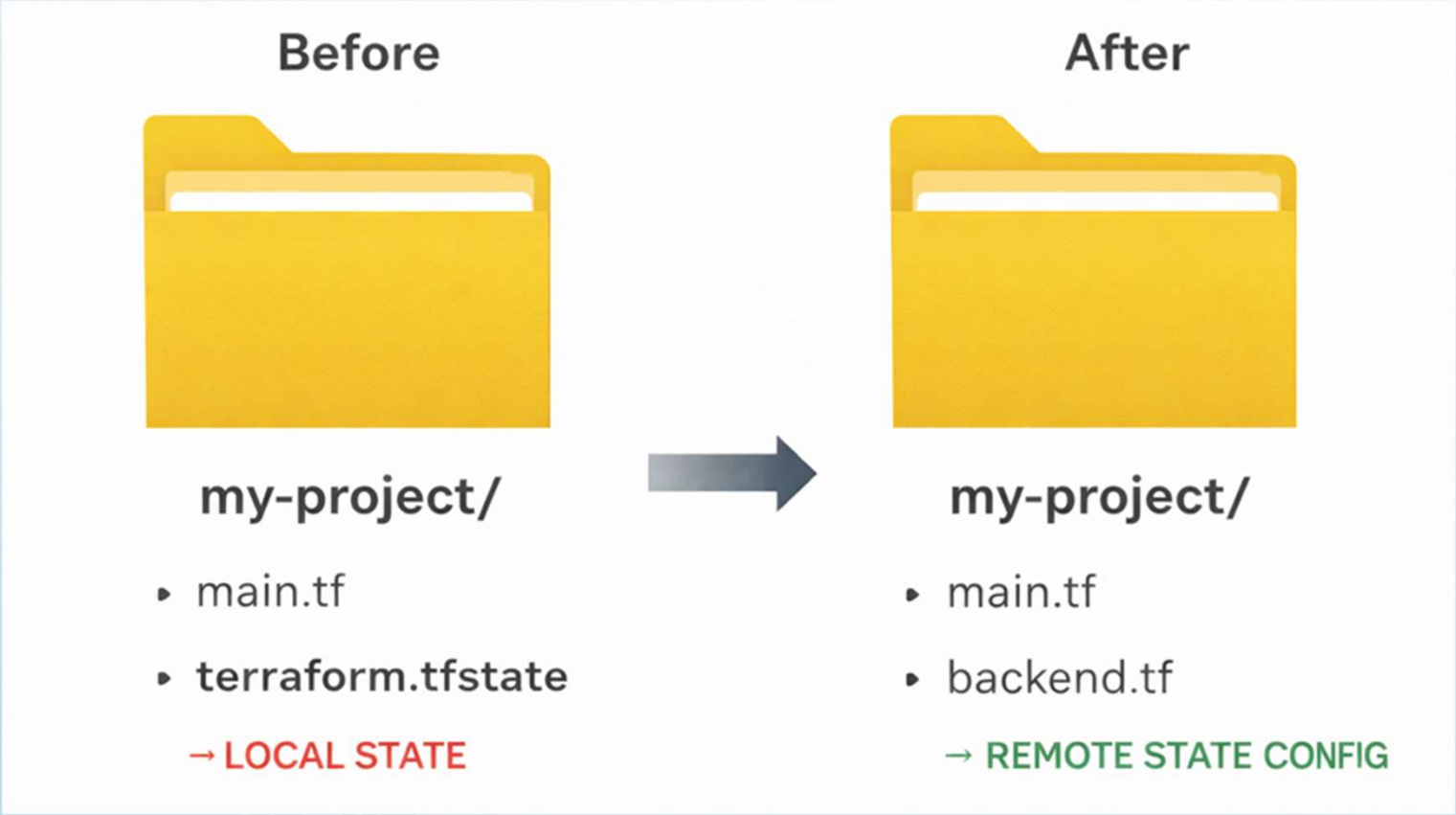
If a lock is stuck:

terraform force-unlock <LOCK_ID>

⚠️ **Warning:**

- Use only if absolutely certain
- Can corrupt state
- Better to wait for lock timeout

Complete Migration Example (Before vs After)



Migration Verification

If a lock is stuck:

terraform force-unlock <LOCK_ID>

⚠️ **Warning:**

- Use only if absolutely certain
- Can corrupt state
- Better to wait for lock timeout